

Advanced Architectures

Using Message Queues For Multiprocessor applications

Introduction

Context switching in real time systems can often absorb large percentages of the available processing bandwidth robbing the system of valuable resources, time and power. However, systems can be architected that avoid these overheads and release significant, and often the entire, context switching overhead back into the available processing bandwidth available for data manipulation. These systems can be built without the need for a Real Time Operating Systems and often can be built without the need for any context switching between processes.

Real time systems often have no hard requirement for preemptive multi-tasking and can and must perform all tasks within a given time-slot. Provided all tasks can be run to completion within the allotted time period no preemptive actions need be taken. Multiprocessor systems can be built that allow for communication between processes by the use of high speed messaging queues that provide, by their very nature, atomic operation. There are often tasks required in these systems that “break the rules” but these are most often off the high speed path through a system and can be handled as slower tasks that may be suspended if required. This allows any required context switching to be performed only on non-time-critical tasks.

Background

The primary problem is how to deal with interrupts. It is interrupts that initiate context switches in the first place. If they can be avoided then so can the context switches they invoke. The primary use of interrupts is to signal to the system that some external event has occurred and that that event is requesting that the processor take some action. The Advanced Architectures’ solution is to provide message queues that allow external events to send a message to the processor by way of placing a

Advanced Architectures

message on the end of a queue. The processor when it finishes its current task will scan its message queues in priority order and deal with those messages, which may invoke a new process. As the old process is finished there is no relevant context to be saved and as the process to be invoked is currently idle, as it previously ran to completion, there is no existing context to be restored only the normal process start-up; usually simply loading the PC with the start of the process.

Interrupts still live in such systems to alert that one or more of the queues is about to overflow or that a queue is going non-empty. Queue overflows indicate an imbalance in the system and that the flow of data through the system is erratic. This usually requires a re-balancing of the system as a consumer task is slower than a producer task. A queue going non-empty is used to indicate a start-up condition whereby currently idle processes can be initiated. This is often due to a break of the flow of data into the system. Here the use of an interrupt is only for a low frequency event and is not considered a part of the data-path flow for the normal processing of data.

Queue based architectures

Queue based system use a data flow model as opposed to the event driven model used by interrupt based system. Systolic arrays of processes move data through the system and balance the processing requirements of each stage in such a way as to ensure that data manipulation is completed within the required latency of the system under development.

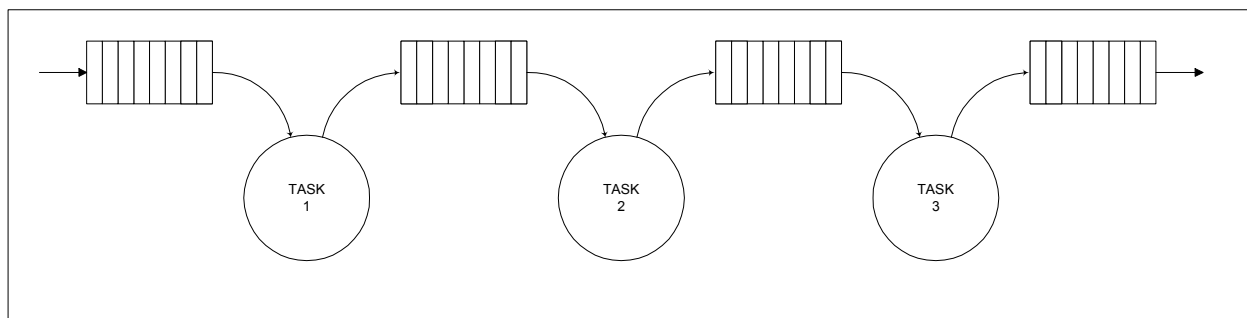


Figure 0-1: Simple task structure

Figure 1 illustrates an example of three sequential tasks passing messages through queues. The queues are used only for passing pointers to real messages that are stored in a memory visible to both tasks. The queues should be viewed as “job” queues with the details of each job specified in a message structure pointed to by the entry in the queue itself. The first task receives messages from an input device that indicates that there is data ready to be processed. The input device pushes a pointer to where it put the data in main memory, onto the queue. This takes the place of an

Advanced Architectures

input interrupt. The first task upon completion of its current job will check the input queue and if it is non-empty will take the next job from the queue and continue. When it finishes a task and puts result data into memory it will push the queue between it and the next task with a pointer to its result data, which then becomes the input data of the following task. This mechanism continues down the line until the last task writes system output data to memory and signals the output device again with a pointer to the data to be output.

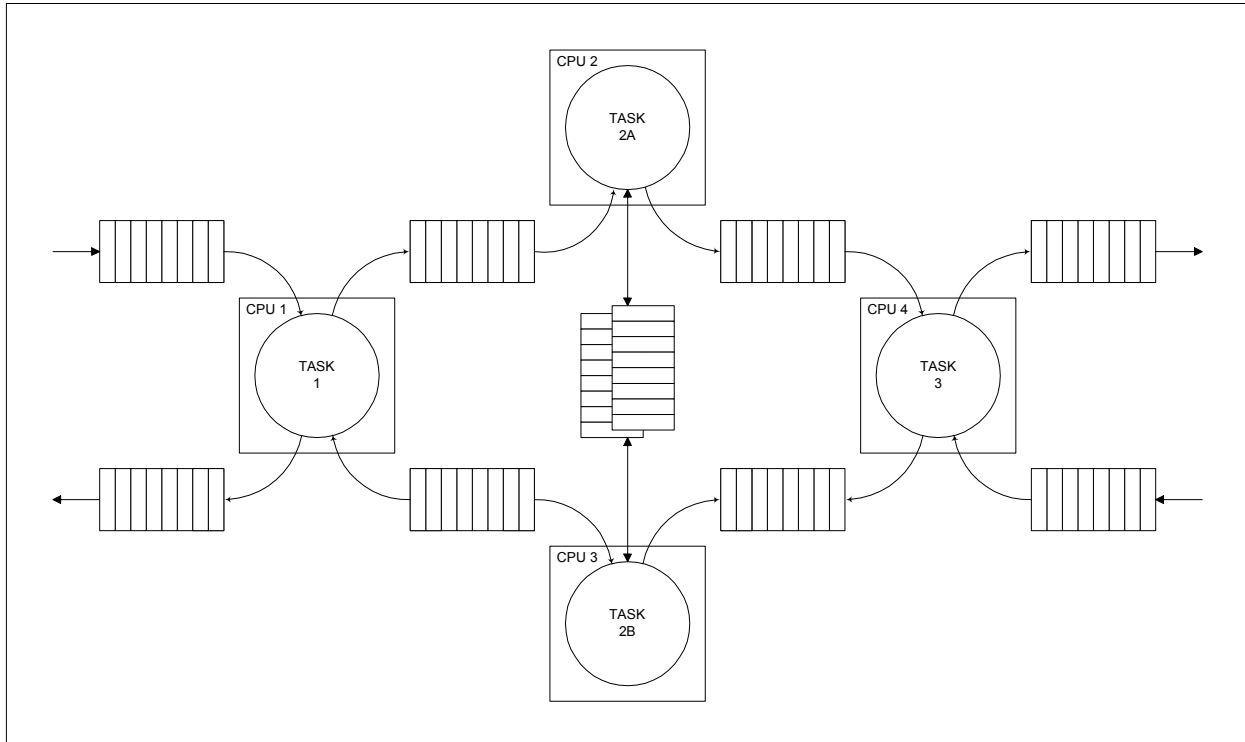


Figure 2: Complex set of tasks on Four CPUs

Figure 2 illustrates that the tasks may be split across multiple CPUs and shows the possible queues between them. In this example optimal performance is achieved when the latency of each task is closely similar and the queues will tend to be half full. Note that task 2 is split into task 2A and task 2B. This is necessary when a task cannot be handled in the “pipeline” latency time and must be split in order to balance the pipe. Queues between the split pair may be necessary if communication is required between them.

The architecture of the system is not fixed and many other layouts are possible. Also many software designers prefer to use linked lists to create their input queues and at first see this structure as restrictive. However, the queues can merely represent the inputs to the linked-list manager that becomes a layer between the task and queue structures; the events arrive at the linked-list manager in time order anyway so the

Advanced Architectures

queues are just latency and elasticity buffers. Figure 3 illustrates an example where the software designers desired to have only one input queue per task.

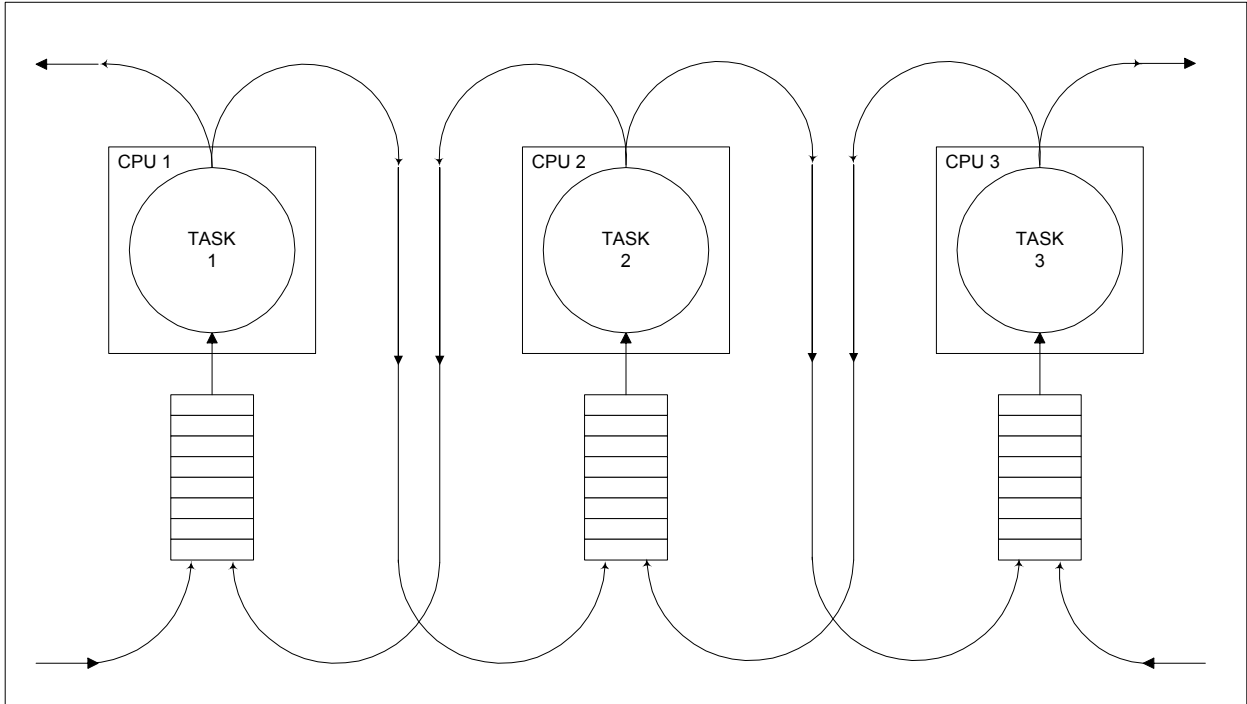


Figure 3: One queue per task.

This illustration shows one task per CPU but the architecture applies to multiple tasks per CPU as shown in figure 4

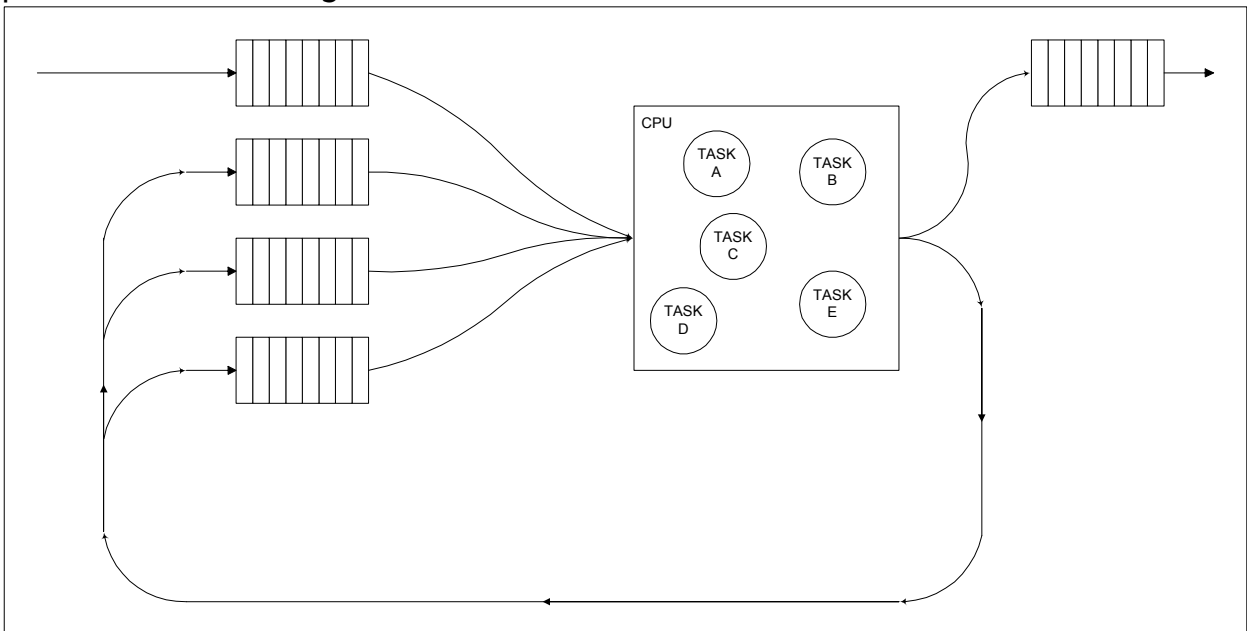


Figure 4: Multiple tasks per CPU

Advanced Architectures

many microcontrollers and RISC CPUs and must either be added or system designers must resort to a hardware solution.

In order to accelerate the queues they may be built in hardware structures in varying levels of performance. By far the fastest is to use a dedicated FIFO for each queue required in the system. This can be useful both in small systems that have little or no shared memory and require ultimate speed. Figure 6 illustrates such a system built using a dedicated RAM as the message memory.

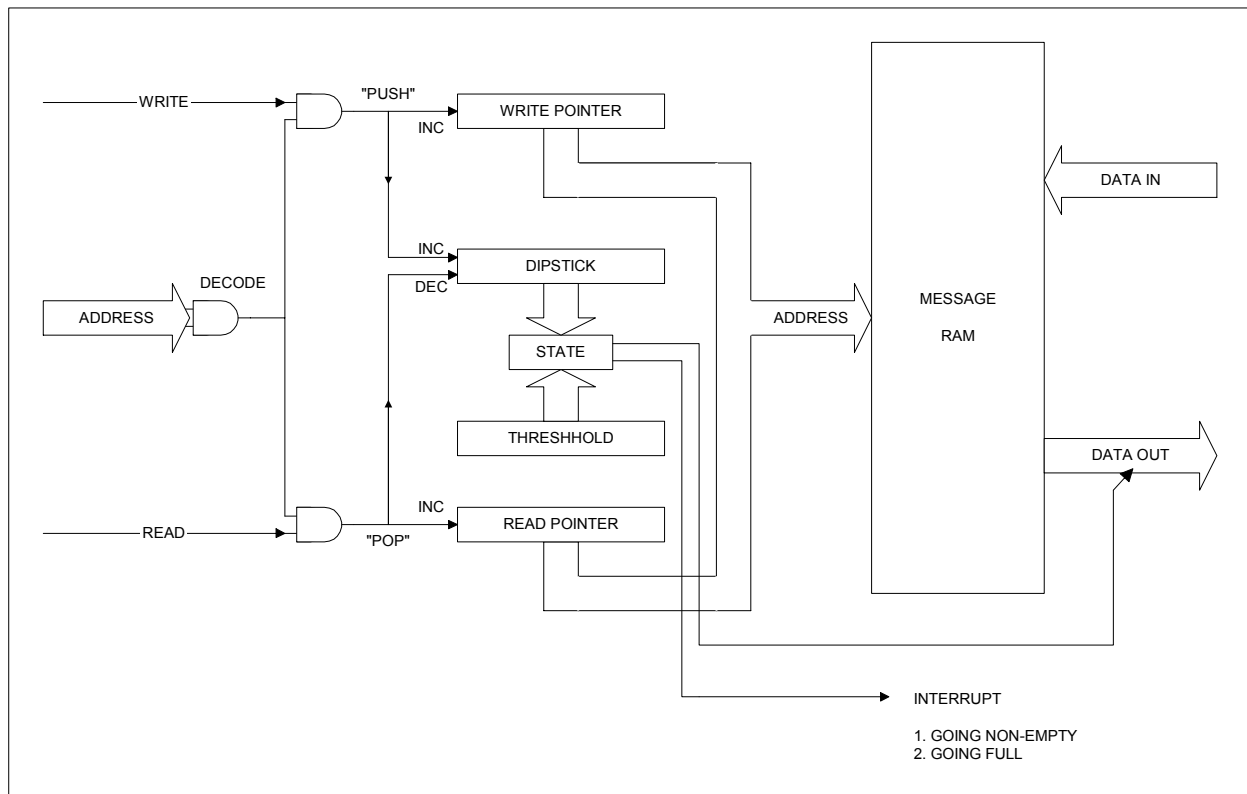


Figure 6: Simple hardware message queue

This figure shows that reads and writes to a specific system address are recoded to mean pop and push respectively. Pushes of data to the queue update a write pointer and pops update a read pointer. A dipstick register maintains the number of elements in the queue and is affected by both pushes and pops. This value is compared to a Threshold register and the result can be used as a status indication. In the above example all messages are assumed to be word pointers to data blocks or structures in another memory. As such the lower two bits are always zero. This system overlays two bits of status information onto these least significant bits to provide the CPU with immediate status information. Figure 7 illustrates the layout and meaning of this data.

Advanced Architectures

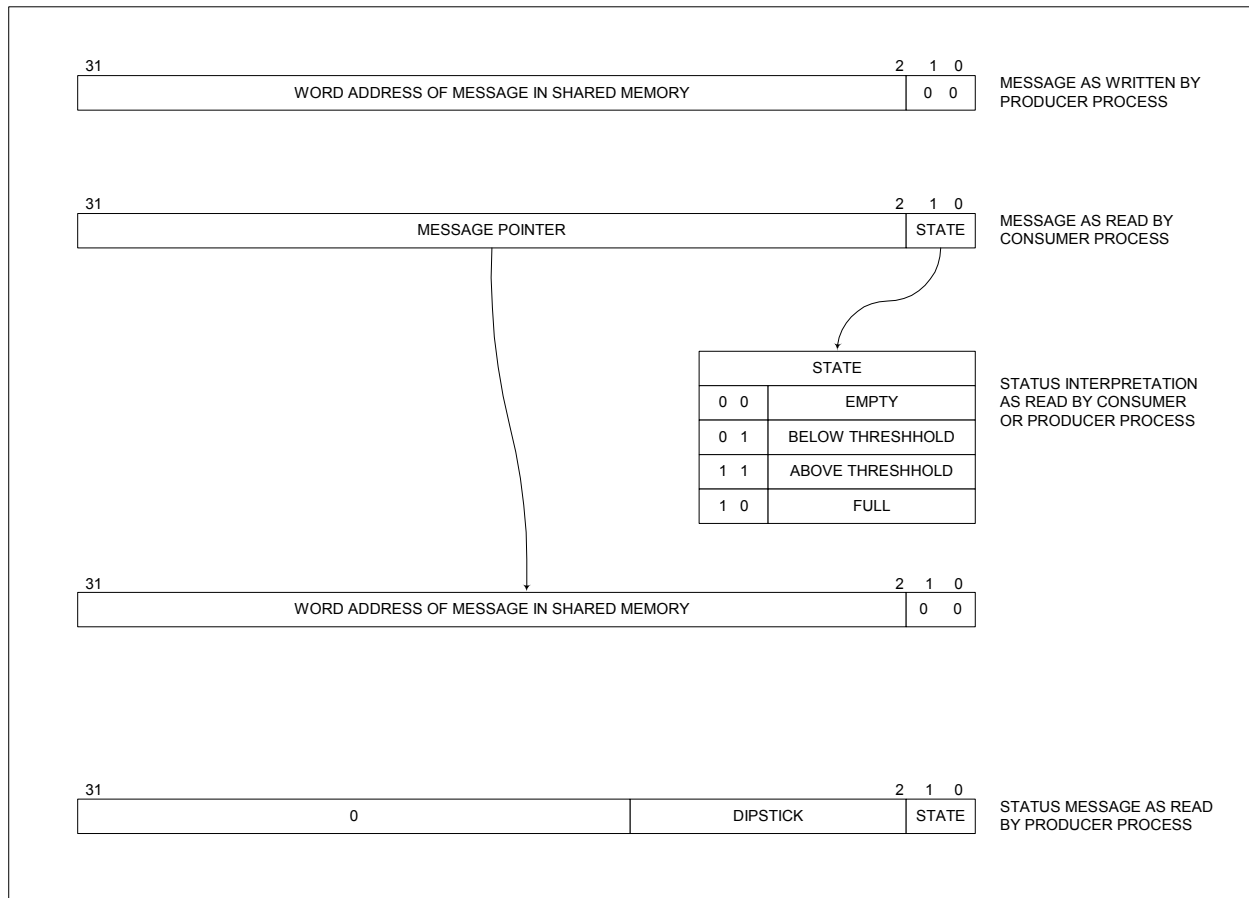


Figure 7: Structure of messages in the queuing system

Producer processes simply write to the queues and so have no returned status information. This requires that they have another address from which they may read the dipstick to indicate the status. The encoded state is also passed on the lower two bits as this often abrogates further processing. Consider a process that needs to send 10 messages and knows that the threshold is set to 50 below the full level of the queue. By examining the status bits the process knows that if the status indicates that the queue is below the threshold that it may safely add 10 messages to the queue without further examination of the queue dipstick. Consumer processes simply pop messages from the queue and check the queue for empty; and finish. The queue is designed such that pops to an empty queue does not further advance the read pointer so underflow is not a problem. This system also generates an interrupt when the queue first goes non-empty to awaken suspended consumer processes and also when a queue first goes full to suspend a producer process or at least tell it to back-off.

A more generalized version of this system can be built that maintains the pointers, dipsticks and thresholds of a number of queues in a register file and then points into the main memory structures of a system. Figure 8 illustrates such a system. Here a

Advanced Architectures

file is kept of control structures for the queues that all reside in a combined message RAM. Every queue is assumed to be of the same size and its address, modified to allow for the queue size is used to produce an offset into the RAM. The update logic is the same as for Figure 7 and only one set is required as only one queue can be accessed per clock cycle.

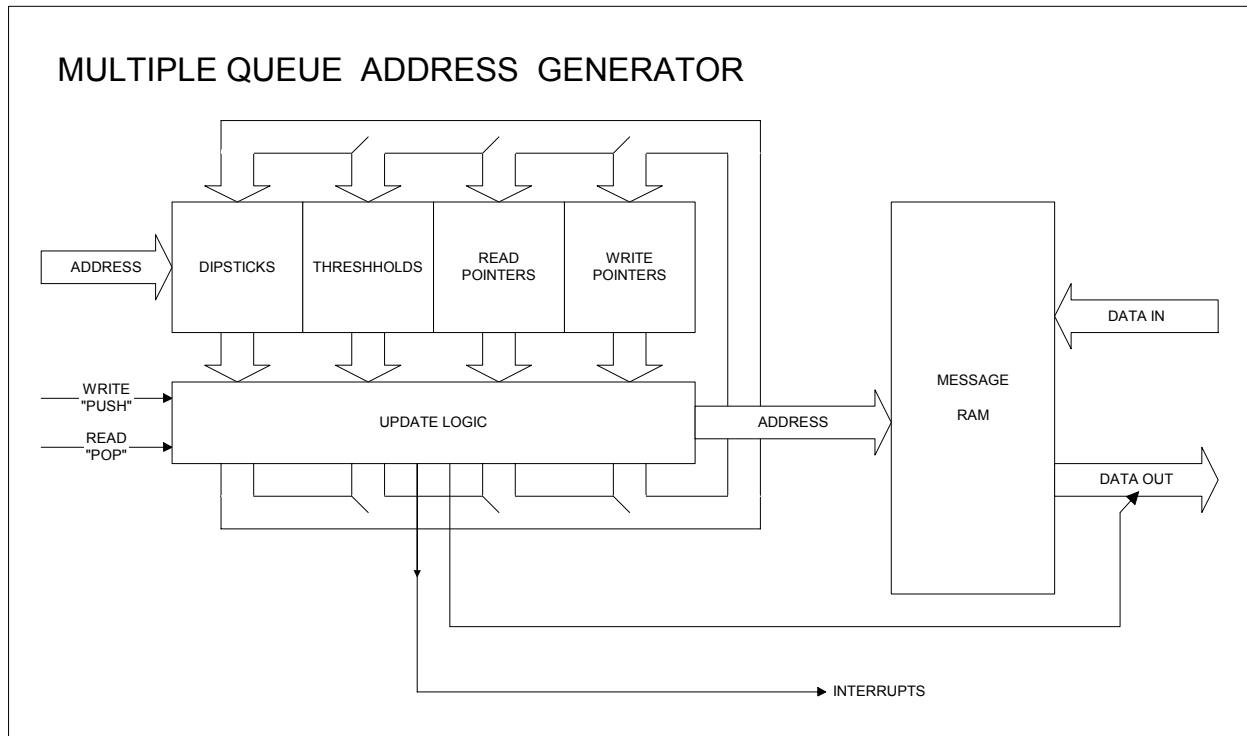


Figure 8: Generalized multiple queue mechanism

It is possible to go even further and build the control structures into main memory and the queue accesses become much more like modified indirect accesses to main memory.

Figure 9 shows a system that uses a shared memory block between CPUs and I/O devices in a chain and each memory block has a queue address generator attached to it. For a number of dedicated systems this provides sufficient flexibility and power but often a more general approach is appropriate and a central mechanism is provided so that queues may be shared between multiple CPUs and I/O devices and a very flexible and high performance system can be built. Figure 10 shows such architecture.

Advanced Architectures

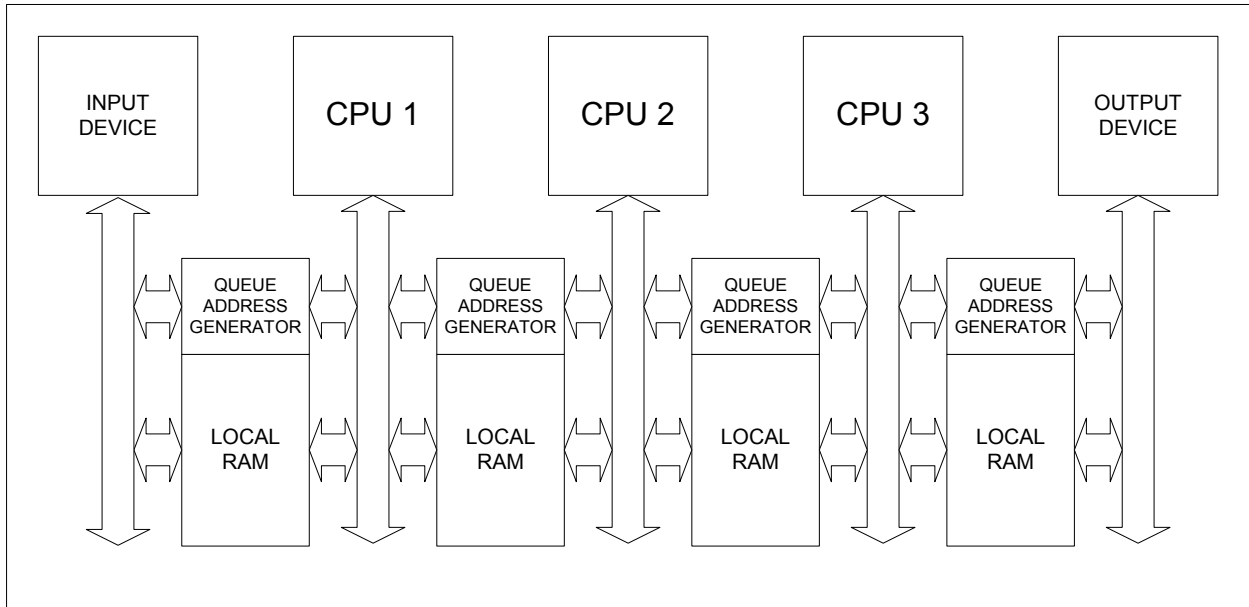


Figure 9 Cascade of queues

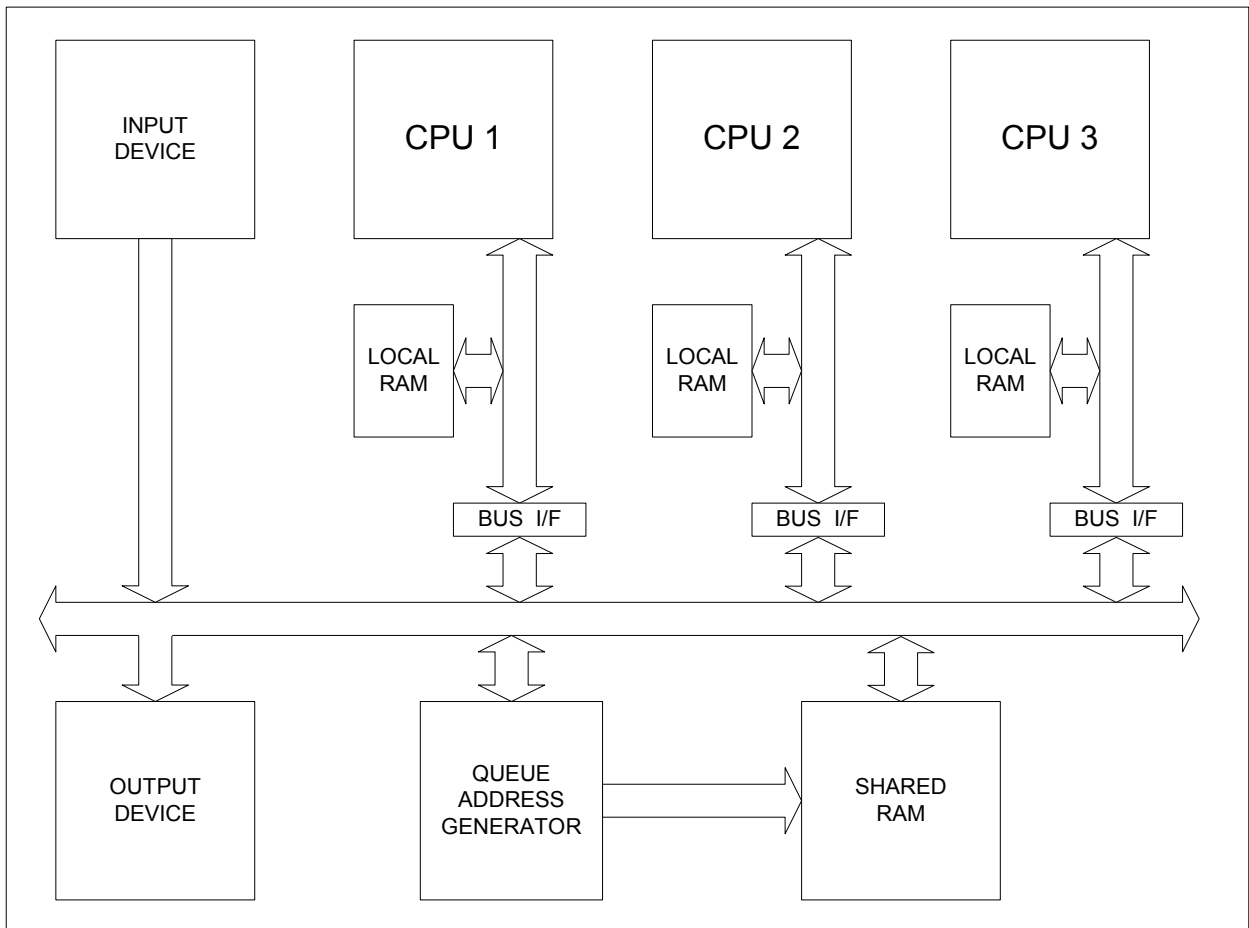


Figure 10: Generalized architecture with central queuing

Advanced Architectures

Conclusion

Advanced Architectures has developed techniques that allow high performance, real time systems to be built that do not rely on context switching in the high-speed portions of their designs. Such systems retain the flexibility of context switching as needed for lower performance sections of the systems developed. Our current customer base has successfully implemented these types of architectures and Advanced Architectures can help develop these systems into complete and successful architectures.

Advanced Architectures

About Advanced Architectures

Advanced Architectures (A2) designs high-performance computer systems, subsystems and components. A2's strengths include the conceptual design and architecture of complete systems including custom DSPs and CPUs. Advanced Architectures is unique as a design house by being able to provide a leadership role in the development of complete systems from concept through manufacturing. A2's proficient interfacing with sales, marketing, manufacturing, and finance ensures corporate success. A2's design experience includes implementing structured design methodologies, performing HDL modeling simulation, performance analysis, logical design, packaging and detailed implementation.

Advanced Architectures has extensive knowledge and experience in the development and application of high performance computing systems. Designs have ranged from complete supercomputer designs to customized DSP solutions with microprocessor based control and custom compute engines for compression and decompression algorithms.

Advanced Architectures is an ARC Certified Design Center and have completed several ARC based systems and are in the final stages of the development of SIMD extensions to the ARC that we have named A2MP. These extensions provide a framework for SIMD processing within the ARC environment, providing memory addressing and other infrastructure and yet still retains the flexibility of the ARC in that custom instructions and functions can be added to the A2MP to tailor the system to an application.

Advanced Architectures also offers the A2B and A2R on-chip buses to ease the development of System-on-chip products. The A2B is a high-speed synchronous system-interconnect structure that is configurable to provide any bandwidth requirements required on-chip. Its protocols ensure optimum bandwidth usage and can sustain bandwidth at peak rates. The interfaces to the bus are similar to the standards BVCI, AVCI and OCP and wrappers can be readily constructed to create compatibility with any or all of the standards. The A2R is a general-purpose register bus intended to provide access to all on-chip registers via a wiring efficient mechanism.

Disclaimer: Information furnished by Advanced Architectures is believed accurate and reliable. Advanced Architectures reserves the right to change specifications detailed in this Application Note at any time, without notice, in order to improve reliability, function or design and assumes no responsibility for any errors within this document. Advanced Architectures does not make any commitment to update this information. Advanced Architectures assumes no obligation to correct errors contained herein or to advise any user of this text of any correction, if such be made, nor does Advanced Architectures assume responsibility for the function of un-described features or parameters.

No license is granted by implication or otherwise under any patent or patent rights of Advanced Architectures or ARC Cores Ltd.

Copyright © 2001, Advanced Architectures.

Advanced Architectures contact info:

www.a-2.com

Advanced Architectures
19421 Sierra Lago Road
Irvine CA 92612-3812
+1 949 412 3486

info@a-2.com